Documentation for Zn.h and Zn.c

Steven Andrews, © 2003
See the document “LibDoc” for general information about this and other libraries.

```
#include <stdio.h>
#include <stdlib.h>
#define allocZV(n) ((int *) calloc(n,sizeof(int)))
#define freeZV(a) free(a)

int *setstdZV(int *c,int n,int k);
int *printZV(int *c,int n);
int *fprintZV(FILE  *stream,int *c,int n);
int minZV(int *a,int n);
int maxZV(int *a,int n);
int *copyZV(int *a,int *c,int n);
int *sumZV(float ax,int *a,float bx,int *b,int *c,int n);
int *deriv1ZV(int *a,int *c,int n);
int *deriv2ZV(int *a,int *c,int n);
int productZV(int *a,int n);
int intfindZV(int *a,int n,int i);
int indx2addZV(int *indx,int *dim,int rank);
int *add2indxZV(int add,int *indx,int *dim,int rank);
int nextaddZV(int add,int *indx1,int *indx2,int *dim,int rank);
int indx2add3ZV(int *indx,int rank);
int *add2indx3ZV(int add,int *indx,int rank);
int neighborZV(int *indx,int *c,int *dim,int rank,int type,int *wrap,int *mid);
```

Requires: `<stdio.h>`, `<stdlib.h>`, `"random.h"`

Example program: `none`

Written 10/01; some testing. Written on Linux, copied to Macintosh. Added tensor routines 10/31/01. Added `fprintZV` and `intfindZV` 3/03. Modified `neighborZV` and added `add2indx3ZV` and `indx2add3ZV` 7/7/03. Added `nextaddZV` 7/14/03.

This library is comparable to the Rn.c and Cn.c libraries, but is designed for vectors of integers rather than real or complex numbers. There are also some routines for keeping track of the indicies of tensors, routines which can also be used for converting the base of a number. For all routines where they are used, `a`, `b`, and `c` are pointers to arrays of integers where `c` is the output array, and `n` is the number of elements (numbered from 0 to `n-1`). Unless otherwise specified, the functions return `c` to allow easy concatenation of routines. Routines don’t have internal error checking and assume the input is good (all vectors defined and `n≥1`).

Functions

`setstdZV` initializes a vector, where the initial value depends on `k`. `k=0` yields all 0’s, `k=1` yields all 1’s, `k<0` yields all zeros except for the element at position `-k` which is 1, `k=2` yields sequential numbering from 0 to `n-1`, and `k=3` yields either a 0 or 1 for each position, chosen randomly with equal probability. It’s not defined for `k>3`.

`printZV` prints the vector contents in a single row of text.

`fprintZV` is identical to `printZV`, but prints to filestream `stream`.

`minZV` returns the smallest value in the vector.

`maxZV` returns the largest value in the vector.

`copyZV` copies a vector.

`deriv1ZV` returns the first derivative of a vector, assuming unit spacing. `n` must be at least 3. See documentation for Rn.c for more details. The function requires a division by two for the calculation of each element of c; as these are vectors of integers and integer arithmetic truncates any fractional part, this means that the derivative will underestimate the actual derivative by about 0.5, on average. The derivative of 0,1,2,3 is 1,1,1,1 as it should be, but the derivative of 0,0,1,1 is 0,0,0,0. The second derivative function has no division and so does not have round-off problems.

`deriv2ZV` returns the second derivative of a vector, assuming unit spacing. `n` must be at least 3. See documentation for Rn.c for more details. The function has no division and so does not have round-off problems.

`productZV` calculates the product of all the elements of `a`. It's mostly useful for determining the maximum address of a tensor or the largest number expressible in a certain base, both of which are described below.

`intfindZV` looks for the value `i` in the vector `a`, which has size `n`, returning the index where `i` is found if it is found, and `-1` if it is not found.

`indx2addZV` converts the index of a tensor element to its address, assuming values are numbered sequentially. `indx` and `dim` have `rank` elements each, where `indx` is the index and `dim` is the size of each tensor dimension. Negative values are permitted in `indx`, which can be useful for determining relative addresses, although all values of `dim` should be ≥1. This does not check if the address is less than 0 or greater than the tensor size. See discussion below.

`add2indxZV` converts an address to the index. The address must be positive. This does not check if the input address is possible. See discussion below.

`nextaddZV` inputs an address as `add` and two indicies as `indx1` and `indx2`, as well as the tensor dimensionality and rank in `dim` and `rank`, respectively. The return value is the next sequential address which is within the sub-tensor defined with the lower corner at `indx1` and upper corner at `indx2`. This is particularly useful in a for loop, in which it is desirable to look at all addresses in a certain region of the tensor. For example,

```
add1=indx2addZV(indx1,dim,rank);
add2=indx2addZV(indx2,dim,rank);
for(add=add1;add<=add2;add=nextaddZV(add,indx1,indx2,dim,rank));
```

This function assumes that every element of `indx1` is less than or equal to the respective element of `indx2`, and it does not account for periodic boundaries. The return value is undefined if add is not within the range defined by indx1 and indx2. If add corresponds to indx2, the returned value is add+1.

indx2add3ZV is identical to indx2addZV, except that all dim elements are set equal 3.

add2indx3ZV is identical to add2indxZV, except that all dim elements are set equal 3.

neighborZV returns the number of neighbors of an element along with their addresses, sent back in c. A return value of –1 means that temporary memory could not be allocated. Otherwise there is no error checking. indx is the element and dim and rank refer to the tensor. mid is used by the function to return the mid-point of c; the neighbors from c[0] to c[mid-1] logically precede the indexed element, whereas c[mid] to c[n-1] logically follow the indexed element. mid may be sent in as NULL if this information is not wanted. There are several allowable type values:

type	neighbor	wrap-around	order	size of c	size of wrap
–1	This code is used to free any temporary memory that was allocated				
0	nearest	none	increasing	≤ 2*rank	N/A
1	nearest	all	none	≤ 2*rank	N/A
2	all	none	increasing	≤ 3^rank-1	N/A
3	all	all	none	≤ 3^rank-1	N/A
4	nearest	partial	none	≤ 2*rank	rank
5	all	partial	none	≤ 3^rank-1	rank
6	nearest	partial	none	≤ 2*rank	2*rank
7	all	partial	none	≤ 3^rank-1	3^rank-1

Nearest neighbors are just those above, below, left, right, etc. of an element; they share a side. All neighbors includes also diagonal and corner neighbors. Wrap-around refers to the neighbors of an edge element, which can be either nothing on the edge side, or the wrapped around element. The non-wrap-around routines return c in increasing order of addresses. For types 0 to 5, addresses are not repeated in c, even if a neighbor is a neighbor in multiple ways, although they are for types 6 and 7. The size of c needs to be allocated beforehand to be large enough for the result. Partial wrap-around means that some dimensions wrap-around, whereas others don't; this information is sent to the routine in the first rank elements of wrap, where a 0 means don't wrap and a 1 means wrap. For types 6 and 7, the information is overwritten with the wrap code, which tells in what way each neighbor is a neighbor. In the wrap code, pairs of bits are associated with each dimension (low order bits with low dimension), with the bits equal to 00 for no wrapping in that dimension, 01 for wrapping towards the low side, and 10 for wrapping towards the high side. See the example below.

Functions types 2, 3, 5, 6, and 7 require small amounts of temporary memory as scratch space. This is allocated the first time the function is called, but is not freed afterwards, allowing the function to be called multiple times with minimal run-time overhead. At the end, it is proper to call the function one more time, with a type value of –1, to free any memory that was allocated. In this case, all other parameters are ignored.

Discussion of tensor routines

The tensor routines assume a tensor is stored sequentially in memory with the 0'th dimension varying most slowly, and the *rank*–1'th dimension varying most quickly. For example, a matrix with m rows and n columns (see Rn.c), which is stored with values filling one row before starting the next row, would be interpreted as having m as the 0'th dimension and n as the first dimension. In that case, the address of index i,j is $a=n*i+j$ and the index of address a is $i=a/n$, $j=a\%n$, where / represents integer arithmetic and % is the modulus operator. These routines can also be used to convert a number from one base to another. In this case all the dimension values are set to the base number, the rank is the number of digits, and the 0'th digit is the most significant digit.

Here are some examples of converting addresses to indicies, indicies to addresses, and of various ways of determining neighbors, using a rank 2 array. The numbers in the array are the addresses, while the row and column numbers are the indicies.

$$\begin{array}{cc|cccc} & & \multicolumn{4}{c}{\texttt{dim[1]=4}} \\ & & \underline{0} & \underline{1} & \underline{2} & \underline{3} \\ & 0 & 0 & 1 & 2 & 3 \\ \texttt{dim[0]=3} & 1 & 4 & 5 & 6 & 7 \\ & 2 & 8 & 9 & 10 & 11 \end{array}$$

Here, `dim` is a vector equal to `dim`=[3,4] and `rank`=2.
If `indx`=[1,2], then `indx2addZV(indx,dim,rank)` returns 6.
If `add`=6, then `add2indxZV(add,indx,dim,rank)` returns `indx`=[1,2].
If `indx`=[2,0], `neighborZV(indx,c,dim,rank,type,wrap)` returns:

> `type`=0 and `wrap=NULL`: `n`=2, `mid`=1, `c`=[4,9].
> `type`=1 and `wrap=NULL`: `n`=4, `mid`=2, `c`=[4,11,9,0].
> `type`=2 and `wrap=NULL`: `n`=3, `mid`=2, `c`=[4,5,9].
> `type`=3 and `wrap=NULL`: `n`=8, `mid`=4, `c`=[7,4,5,11,9,3,0,1].
> `type`=4 and `wrap`=[1,0]: `n`=3, `mid`=1, `c`=[4,9,0].
> `type`=5 and `wrap`=[1,0]: `n`=5, `mid`=2, `c`=[4,5,9,0,1].
> `type`=6 and `wrap`=[1,0]: `n`=3, `mid`=1, `c`=[4,9,0], `wrap`=[0,0,2].
> `type`=7 and `wrap`=[1,0]: `n`=5, `mid`=2, `c`=[4,5,9,0,1], `wrap`=[0,0,0,2,2].
> `type`=6 and `wrap`=[1,1]: `n`=4, `mid`=2, `c`=[4,11,9,0], `wrap`=[0,4,0,2].
> `type`=7 and `wrap`=[1,1]: `n`=8, `mid`=4, `c`=[7,4,5,11,9,3,0,1], `wrap`=[4,0,0,4,0,6,2,2].

Here is a fragement of code for determining the type of wrap-around, using one of the last two forms of the the `neighborZV` function:

```
for(j=0;j<rank;j++)
  wptype[j]=wrap[i]>>2*j&3;
```

The result is 0, 1, or 2 on each dimension, where 0 indicates no wrap-around, 1 indicates wrap-around towards the low side, and 2 indicates wrap-around towards the high side.